Complish

November 11, 2015

Goals of the Language.

Complish is a programming language whose primary purpose is English-like syntax. Its identifiers may contain spaces and must pass a spell-check. It efficiently re-uses short common words via polymorphism, multimethods, namespaces, tenses, inflections, determiners, and multi-use functions. It supports user-defined units of measurement including subclassing basic types like Text and Number. It prefers declaratively stating constraints between entities so it can automatically discover contradictions. It supports temporal connectives and named durations of time in order to make sequencing explicit and checkable at compile-time. It performs higher order functions with gerunds and function invocations with relative clauses. It distinguishes bound parameters from unbound with articles, and lists from singulars through determiners. It is case-insensitive and light on punctuation, and comes pre-loaded with the knowledge of how to properly conjugate some 17,000 verbs. Finally, it unifies objects with functions by way of relations to check user-created constraints at compile-time.

Storage versus Computation.

Functions and objects/structs are treated as similar things: an N-way relation between nouns. There's little syntactic difference between defining a computed relation (function) and a stored relation (struct/object). So just as you can have a mostly-complete instance of a struct and ask what should the remaining piece be – searching through all instances – you can also pass all but one parameter to a function and get the missing parameter returned.

Relations are defined with a complete sentence, in either imperative or declarative mood.

> To give X to Y
> P gives X to Y
> Give X to Y

At the end of the declaration is punctuation to end the sentence – which defines the relation as a struct – or punctuation such as commas or colons to indicate more words follow for defining the function body.

> To give X to Y, <list of instructions>.
> P gives X to Y.

The infinitive definition lends itself well to defining a computation, while the declarative lends itself well to storage, but it's how the initial, main clause of the sentence ends that decides computation versus storage, not the form of the clause itself.

Throughout this manual, when we refer to a relation we are referring to both structs as well as functions. Effectively, this encompasses anything with a verb in it.

Types.

Ordinals, Boolean and plain old numbers are kinds of enum, though the names of their members may be partially computed rather than listed in full somewhere. Similar are the variations on number, called units, of which money and percent are fine examples. Defining new enums is similar to defining a new struct relation with comma-separated parameters, except for the conjunction used is or rather than and: A make is Chevrolet, Ford, or Lamborghini. The word either is optional since or is always exclusive.

time uses the same plug-in system of printing/parsing that user-defined types use. Almost relatedly is duration, which is a named, elastic measure usually defined by two events (one marking the

beginning and one marking the end), but absolute measures of time could be an option, such as three minutes after an event began the duration. Durations help define when a constraint rule should apply.

Lists appear in English as the determiners MANY, SOME, and SEVERAL. Indexes into lists, if ever used, are of course of type ORDINAL.

The two relations which define object-oriented programming are IS-A and HAS-A. The former is known to Complish as A-IS-A ("A cat is an animal.") The latter is known as A-IS-A-AND ("A CAT IS ") as well as OF-A ("the cost of the car").

PROPERTY is another basic type, a category of similar instance-to-instance relations invoked with X OF. The X is part of the relation's name just like the OF is – such as COST OF. The relation relates two types – such as an enum called "car" to a number like MONEY. And invoking the relation's name on the former returns the latter's instance – THE COST OF THE 1978 CHEVROLET MUSTANG returns a specific dollar value.

An OBJECT is an aggregate type, defined by "is" and named per the sentence's subject: A CAR IS A MAKE, A MODEL, AND A YEAR creates a noun, "car". An object defined by a main verb is named by the verb if anything: A PERSON LOVES ANOTHER PERSON creates a relation, "loves", with noun form "loving". Classes are enums used in several property relations, so aren't per-se a base type of the language. Rather, the property relation works in tandem with other relations, IS-a and HAS-a, to give us our familiar hierarchy. The compiler has rules regarding these: IS cannot be circular, for example. Struct is synonymous with object because functions are multimethods, not methods.

Type TYPE is intended as parameter only. All parameters of type TYPE are type-parameters, not actual parameters. This is how generics work.

Constraints aren't first-class types.

Articles Define and Name Parameters to Functions.

When defining a relation, the relation's base name is always whatever verb you'd like to put into the sentence. The nouns follow stricter rules, which you might call naming conventions, but in truth the parameters are automatically named: it's types that appear in the main clause. This keeps code readable.

- A function is a verb, and the nouns that go with it are its parameters. Excepting the subject and direct object which are placed immediately adjacent to the verb, a preposition heralds another noun belonging to the function. English has over 200 prepositions. (Prepositional phrases must follow the direct object, never precede the subject?)

- Each parameter-noun must have its type specified, but not necessarily a name.

- In the commonest case, we use A, AN, or SOME in front of a singular noun, but not THE. The singular noun that follows the article is the type, like car or person: A CAR; A PERSON; A NUMBER. Mass nouns read differently, and so we have SOME MONEY or SOME WATER. Some are interchangeable so use whichever you are most comfortable, such as A TIME or SOME TIME. These articles are all synonymous so use what reads best.

- Parameters are later referred to within a function body by THE followed by the type: THE CAR; THE PERSON; THE NUMBER. If there are two or more of the same type in the main clause, as in A PERSON TRADES A CAR TO A PERSON FOR A CAR, we use the longer form of the names that incorporate the verb and leading preposition. In this example they are, in order: THE PERSON TRADING, THE CAR BEING TRADED, THE PERSON BEING TRADED TO, and THE

CAR BEING TRADED FOR. Those names are certainly long, but in practice functions don't tend to take multiple parameters of the same type unless it's a basic value type, like integers. But for those...

- The commonest value types of a programming language appear so frequently in function signatures that in the interest of brevity they have adjective forms: NUMERIC, TEXTUAL, ORDINAL, BOOLEAN, MONETARY, TEMPORAL, UNIQUE, and SEQUENTIAL. The adjectives precede the nouns, which are the real name. So TO APPEND A TEXTUAL BASE TO A TEXTUAL APPELLATION would have parameters THE BASE and THE APPELLATION, both of type TEXT. (Can this be reversed, as in "a horizontal percent" and "a vertical percent"? Is there an unknown term (one or more unknown words) allowed for every base type parameter? How would it differentiate between a name and a typo?)

- A verbose form allows naming divorced from type. A trailing parenthetical, as in TO TRANSFER A PURCHASE (A CAR) TO A RECEIVER (A PERSON), uses parenthesis as English intended: to provide supplementary information. It is usually only useful in cases where the same class is being passed multiple times and not as a list, such as in A PRIEST (A PERSON) WEDS A GROOM (A PERSON) TO A BRIDE (A PERSON) BY...

- Lists have special syntax. Instead of A, AN, or SOME we use MANY for a list of that noun: MANY CARS; SEVERAL PEOPLE. We can bind the number of elements with AT MOST 5 PEOPLE or AT LEAST 2 CARS. (The use of "at" here doesn't preclude using it to flag a parameter.) Lists are written as in English: items separated by commas, with the conjunction AND or OR appearing just before the final item. The Oxford comma is optional so that lists of only two items do not produce a special case.

- The direct object of a function is IT. Or if a function has only one parameter, it is IT. IT is very short to type. Curried functions can use IT concisely because they only have a single input and a single output.

- One parameter may immediately follow the verb (called the direct object, in English grammar), but after that each parameter needs at least one preposition in front of it: GIVE A CAR TO A PERSON. An exception to this is two nouns immediately following the verb, as in GIVE A PERSON A CAR, which the compiler understands the first as a "to" parameter and the second as the "immediately following" noun, effectively re-writing GIVE X Y as GIVE Y TO X.

- The verb IS looks to break the above rules because it seems to take a comma-delimited list of stuff rather than a "preposition-ed" list of stuff. But in fact it takes only two parameters, subject and direct object, and the direct object is "many properties": AN UNKNOWN NAME IS MANY PROPERTIES VIA....

- The prepositions BY and VIA bend the rules by actually preferring a particular type after itself. It prefers to associate with the type of "function body", and the body of a function spans sentences until a different function is started.

- A parameter's type may include extra adjectives that are the past participles of verbs which were defined elsewhere as functions. So A SANITIZED TEXT would declare the parameter THE TEXT which was recently, or if not will be immediately, ran through the function TO SANITIZE A TEXT. (For multiple participial adjectives, how important is order? And what operations make the text de-sanitized again?)

- All functions are multimethods. Function-selection is based on the polymorphic type of all arguments, not just the first. Since most functions have a single verb as their name, multimethods allow reusing a verb for many purposes without conflict. Selection is based on a simple distance formula, with ties broken by giving left-most arguments slightly greater weight. The dispatcher can call out of the assembly to handle new classes or methods added after compilation, in order to solve the Expression Problem.

- With referential transparency, there's no difference between pass-by-value and pass-by-reference; they yield the same result. ??Parameters are always passed by reference except when forced with the adjective COPIED or the preamble A COPY OF. This is because 1) we shouldn't have to know the size of a type to know how it works, 2) the most interesting parameters are objects anyway, 3) passing by value can always be an optimization, 4) explicit plain-English language exists, even in adjective form, to name and create a copy (but not for a reference?), 5) in everyday programming practice, it's uncommon anyway, because 5a) we don't need the extra four bytes of memory, 5b) using the same variable for two unrelated concepts is always a bad idea, and 5c) code added later may not notice the value was replaced by something else, leading to maintenance headaches.

Functions are Reversible.

Since relations encompass structs and functions, and structs can be used in a fill-in-the-blank method, then it must be true that functions can be used in this way as well: any mixture of parameters may be given to get the answers for the ones not provided. This idea comes from Prolog, a relational language in which functions bind values to variables, only to rollback and re-try bindings when it discovers the first-guess bindings cannot work.

Complish doesn't backtrack. Its functions can be called with any combination of bound and unbound parameters and the function could run, filling in the unbound parameters with return values. The primary example is a parse/print function, which deserializes or serializes depending on whether it was given an object argument or a text argument.

The reason for this odd feature is to support invariant constraints and declarative programming. By declaratively expressing the relationship between an object and its textual representation it should be obvious how parsing and printing are both done. Complish doesn't worry about a function being reversible unless it is actually used that way.

Anyway, using EACH or EVERY on a partially applied function will run through all the returned combinations.

Immutable Variables Hide.

Complish's variables don't seem to be immutable because the value of X at the beginning of the function won't match its value at the end. The reason for that is the X at the end of the function isn't the same variable; it's the same name on a different variable. Remember than a variable has an existence separate from its name. Here, the old immutable variable is still in scope but its name was re-bound to the newer value, while another name, THE X BEFORE YING, was bound to it. Just as functional compilers will optimize away the copying of an immutable variable to a newer version of itself by actually treating it as mutable under the hood, Complish does the same but with the added trick of swapping name bindings: re-binding the original name to the new variable, and binding a new name to the old variable.

Alternately, if a parameter-variable's name has the word BEING in it, it is a name *being* continually re-bound.

Relative Clauses Invoke Functions and Search Structs.

Functions can be invoked with a relative clause of the form THE (type) WHICH. So for the relation GIVE A CAR TO A PERSON, we can use it like THE PERSON WHICH WAS GIVEN (the car) TO to return a person, while THE CAR WHICH WAS GIVEN TO (the person) returns the appropriate car. The relative clause syntax allows asking for any single parameter as output when supplying the other parameters with input. It resembles a typical query of SQL or Prolog. And similar to both, the syntax can return a list rather than just the first value by prepending EACH, EVERY, or ALL THE. Otherwise, relative clauses assume there is only one value to return for a given set of inputs.

Multiple Return Values Can Be "Gone Gotten".

A function can define a new local var which names an intermediate result. The caller can "go get" this value, giving us return values that weren't explicitly passed in or out. The name of the "gotten" value is similar to its name inside the function, but without the continuous tense. So if the full name of the variable inside the callee was THE MONEY BEING TRADED AWAY, it's known to the caller as THE MONEY TRADED AWAY. The usual way of getting a return value is still the relative clause, but for cases where it can't be used this is the way to go.

Imperative sentences, meaning relations which are functions and given all arguments, tend to make a change in some state "elsewhere" rather than returning a value. It looks like it must be changing state because since all arguments are supplied, it has no place to return a value. But it can: any values "gone gotten" by the caller count as a return value, plus any or all of the parameters themselves are return values. Complish avoids the constant need for x=f(x) by assuming x is changed by f(x). So in f(x,y,z) it is assumed that all three could have been changed.

The syntactic structure BEFORE <VERB>ING refers to a variable's value as it was just before that (most recently) mentioned verb. So in the above example, we might want to say we started with THE MONEY BEFORE TRADING. Typically, it is assumed that any or all arguments to a function were changed, so the BEFORE <VERB>ING syntax allows access to the immutable inputs.

;

We can ask for multiple parameters of a single query by using names as well. After the search, the local parameters' names are exported to the caller for use after the invocation. So if we have functions TRADE SOME MONEY FOR A CAR and GIVE A CAR TO A PERSON, in the sequence

> TRADE $50,000 FOR THE CAR WHICH WAS GIVEN TO A PERSON;
> SAY "We have [THE CAR GIVEN] which now belongs to [THE PERSON GIVEN TO].".

This would be how to "go get" any return value, rather than passively accept the one and only return value that a particular function decided to grant. The new name, auto-created and auto-named from the invoked function, points to the callee's copy of it, as if the relative clause's "input" parameter worked like the REF keyword of C#. (Under the hood, it's actually copied from callee to caller, into the stack space that the caller specifically reserved for it.) These are like implicit parameters, but as implicit return values.

(Of course, this only makes sense with mutable variables.)

Pass Functions to Other Functions with Gerunds.

A partially-applied function is a function that has values supplied for only some of its parameters. Although this can be done at compile-time in the plainest of Pascals, languages with explicit support can create such wrappers at runtime. The run-time ability to change a function's signature by wrapping it helps when trying to string together a series of very different functions. Wrappers act as adapters, and partial-application is how to reduce the number of parameters a function needs.

English can use a verb as a noun, with or without the verb's attached nouns. Gerunds and gerundial phrases naturally fit passing functions to functions.

How about an example of an ordinary function taking a function parameter, and using it on some stuff?

- TO LEARN REPAIRING A CAR WITH A TOOL, ... This defines a function that takes a function. Invoking the outer function might be done like, LEARN RE-BELTING THE BUICK WITH YOUR PANTYHOSE. Inside the learn function, what do we wish to do with the parameter? (What's the parameter's name? THE REPAIRING?) We might want to just store the invocation for later, or look up the invocation in a list of same, or actually invoke it. For the former two, a short name to sling around is desirable: REPAIRING, THE REPAIRING, or even REPAIR. Invoking is easy in this case since the parameters come with it: DO THE REPAIR, TRY REPAIRING, REPAIR IT, or even just DO IT. Note that the passed function may have nothing to do with repairing. That's just the name the outer function uses in its body to refer to whatever function was passed to it.

- But what do we do with a case of partially-applied or even wholly unapplied functions? Invoke the above as LEARN RE-BELTING A CAR WITH A TOOL. Now what? Or even LEARN RE-BELTING THE BUICK WITH A TOOL. With the instances of inserting or looking up in a list this partially-applied or wholly-unapplied function, little has changed except the type of the list. (Usually such a list only cares about the number of unapplied arguments, and that each item within has the same types for each unapplied argument. Applied arguments are effectively an assignment statement in the first line of the function body.) Granted, the compiler makes a distinction between functions with zero unapplied arguments and non-zero, because that determines if it can be invoked.

- Lambdas come with some restrictions because of syntax issues. Statements can be put into a list, and lists can be delimited by semicolons as easily as commas. So when defining a lambda join the statements of its body with commas or semicolons instead of periods. The lambda ends at a period. Of course, said period also ends the outer statement, but since arguments in Complish can be rearranged thanks to the prepositions marking which slot is being bound, it's only a problem with statements taking multiple lambdas or relative clauses taking a lambda. These limits are acceptable since a lambda could always be put into a local variable.

- How to begin a lambda? The phrase SUCH THAT works in math and English both. Prepend a parameter list to one like A CAR AND A NUMERIC DAYS SUCH THAT and the rest follows from that, semicolons and all. The parameters are a comma'd list not a preposition'd one.

- There's also "lambda light" cases like EVERY CAR WHOSE YEAR PRECEDES 1974 which acts as a where clause to filter the list. It's a bit shorter than ALL CARS SUCH THAT THE YEAR OF THE CAR PRECEDES 1974. In the middle is EVERY CAR SUCH THAT ITS YEAR PRECEDES 1974, which doesn't merge the possessive with the relative pronoun. Both WHOSE and SUCH THAT set IT to each item in the attached list it's filtering. But SUCH THAT can be easily attached to a comma'd parameter list instead of a concrete list of objects, wherein it doesn't set IT but does create local THE variables.

- To learn a skill, ... Here note that skill seems to have no adjective describing its type as a simple type, so it must be an object, like car, correct? Well, perhaps not. Re-belting a car with a tool is a skill. To re-belt a car with a tool, ... Now skill defines a type, like a C# delegate. But it's an "opt-in" delegate; only those relations explicitly listed as a skill are considered to be one. Rather, we could go the other way, where a skill is any relation accepting a car and a tool. Or, explicit listings could take various signature, and applying nouns to one for invocation could ignore the un-needed arguments. Effectively if re-starting a car were also classed as a skill, then to invoke a skill would require specifying both a car and a tool, even if the function being invoked is re-starting, not re-belting. This would allow skill to participate in inheritance chains with other delegates.

- Zarfian could categorize functions by patterns on name as well as if a variable was read or modified, or a particular section or file contained the function, or by what functions the function called. By providing many little ways of describing functions, Complish can group multimethods concisely and ad-hoc. If such a grouping is named, then it can be a type for a parameter.

- Repairing a car with a tool requires a mechanic. This defines a complex struct, a relation between a function and a noun.

Relations.

The bones of Complish are relations. Verbs are relations between nouns. The preposition via establishes a relation between a definition and the defined. Subordinating conjunctions like before and after are temporal relations between other constructs. But the most basic relation is is. Its several definitions are divvied up thusly:

- The a-is-a relation, which is the familiar is-a of OOP. When two structs are connected by a-is-a, then operations such as allocation, deallocation, and property-lookup affect both as if they were one. A-is-a is a non-circular asymmetric relation. Unlike is-and and is-or, a-is-a does not accept a list of anything, on either side of the verb.

- The the-is-a relation defines the first term to be of a given type. When we say that the appellation is of type text, it's this relation we're referring to: *the* appellation *is a* text. It also defines an instance of storage, though the storage may be in a larger struct, defined elsewhere, or a standalone variable.

- The a-is-and relation defines a struct type. The sentence should begin with a word which doesn't take the first slot of a the-is-a relation, but likely takes the second slot. After the is comes a list of names which do participate in the first slot of the the-is-a relation. The sentence, and this relation, define that initial word to be a struct. Effectively, it defines a larger piece of storage. Note: it is an error for the inital term to have a "the".

- The is-or relation, used by a sentence which begins with a word not in the first slot of an the-is-a relation (but can be in an is-and relation), followed by is and a list of names (typed or untyped). This creates an enum. If the initial word is in an is-and relation, the values of the enum are effectively peer subclasses of the initial word, which is their parent class. The only difference between whether the initial term is "a" or "the" is that the "the" will also create a variable of that type, with that name, which can be assigned to and read. A singleton, in other words.

- The OF relation is the property relation. It is a function on structs, not a struct itself, and returns (a pointer to) a storage location. The right-hand side of the relation is a word which was on the left-hand side of an A-IS-AND definition. The left-hand side of this relation is one of the names listed on the right-hand side of that same A-IS-AND relation. Effectively, the A-IS-AND relation "writes" this relationship while the OF relation "reads" it.

- Take a moment to note the two different is-a relationships, A-IS-A and THE-IS-A. The former specifies object inheritance. The latter specifies an ordinary, typed variable (global, local, property, or otherwise). Neither accept a list on either side of the verb.

- Objects and classes are incidentally created with this system. An A-IS-AND creates a struct. Another A-IS-AND creates another struct. Join the two structs with A-IS-A and the left-hand one inherits the right-hand one. Create an instance of the subclass with THE-IS-A. Read and write the values in the class'es properties with OF.

- The INSTEAD OF relation defines a handler.

- The BY/VIA / [IN-ORDER]-TO-<VERB> relation connects a function signature to its body.

Constraint Satisfaction, not Constraint Solving.

There are three categories of constraints. Unary constraints restrict the domain of a variable. Temporal constraints specify order of operations. The third and most important specifies invariant rules between variables via expressions as a tree of dependencies.

Unary constraints can be preprocessed by altering the details of the variable's type. Temporal constraints merely sort invocations into a chronological ordering. It's the invariant constraints which are the difficult ones. A lot of imperative coding just maintains invariants. "This should always be true" usually requires sprinkling a lot of checks, corrections, remedies, and exceptions all over the place. And sometimes the invariant must be broken in one specific place, because that place is the one telling the rest of the code that the invariant exists.

First of all, Complish is concerned with constraint satisfaction, not constraint solving. Complish requires that most variables have an initial value, which it checks at compile-time to ensure invariant satisfaction from the get-go. Then Complish looks for cases where functions modify variables in a way that eventually violates the invariants. This involves a graph, the nodes of which are typed variables and the arcs are the constraints themselves. Although the constraints in said graph only allow binary constraints, higher-order constraints can be represented by a constellation of binary constraints.

Functions should be viewed not as a list of instructions but as an invariant constraint. Functions can be evaluated for any combination of its parameters to return the unbound parameters in the same manner as struct relations can be searched by any set of values. And so, Complish should be able to calculate the allowed range of every parameter in the source at compile time. For mismatches between functions when one's range doesn't perfectly match the other's, Complish can either reduce the larger range to match the smaller, thereby reducing all its parameters indirectly and setting off a cascade, or complain to the user. Unsolvable equations demand the user define how that case should be handled.

;

Constraints in the Kaleidoscope sense relate a duration of time to a declarative statement like X IS AN INTEGER or Z IS X + Y. Numerical formulae, boolean algebra, regexes, and mapping storage locations to types all can be part of a temporal constraint. A temporal constraint relates a duration

to a (non-temporal) relation. Constraints which don't seem to relate to time relate to ALWAYS. (Or NEXT, in the imperative.)

The before/after rules of Inform and AspectJ trivially fall under this umbrella: relating a named block of code to execute in relation to an inline block of code.

The "precedes" keyword of Zarfian relates a description-of-rules to another description-of-rules, creating two ad-hoc groups of rules with a temporal ordering. Besides a rule specifically invoking another rule from within itself (at which point the callee was called a function), this keyword defines a sequencing of steps besides the metarule of "most specific first".

All imperative blocks of code must be related, directly or indirectly, to a temporal word that hints at when that code runs. If a imperative block is named, the name must be used in another block, pushing back this problem a level. Eventually, the topmost block runs when the program is started, presumably "once" but possibly "always" or even "during" something else going on in the operating system.

Aspects, Events, and Exceptions via Subordinating Conjunctions.

INSTEAD OF is like the override keyword of OOP. It allows defining handlers for exceptions, events, and other such pre-existing bits of code, named by gerund phrases. Further conditions can be hung on it: INSTEAD OF DIVIDING BY ZERO WHILE TRADING AWAY A CAR FOR A CAR, etc.

The subordinating conjunctions BEFORE and AFTER can also be used to implement aspect-oriented programming. Given the multimethod GIVE A CAR TO A PERSON, we can use these conjunctions to attach a rule to the multimethod, using an appropriate form of the verb: BEFORE GIVING A CAR TO A PERSON, etc. The function otherwise acts like any other function.

Conditions can test whether a method is currently executing with WHILE or DURING, again using an appropriate verb form: BEFORE TRADING AWAY A CAR FOR A CAR WHILE/DURING GIVING A CAR TO A PERSON, *etc.*

Immutable Input Files, Self-Consuming Output, and Infinite Realtime.

Videogames use a lot of input files – videos, graphics, sounds, music, text – that is effectively immutable. Assuming that game start checks that the files exist (and maybe the file's general shape isn't broken, and possibly even mark the files readonly at the OS level before verifying contents), then those input files could be considered immutable like the number 5, and hence, easy to use in a FP language without sticking it or "IO" into the input args. If the gameplay needs to load the file at runtime, and the file is missing for whatever reason, the thread will block until it is there.

Output that happens during the execution occurs "in another world" which we do not model. Output functions return a special oroboros value, a recursive lambda that does nothing, which won't get the output statement optimized away because the return value uses itself. Multiple such statements listed together will not have their relative order guaranteed, only that they run. Multiple output statements should be combined into a single statement taking "many somethings". (Wait, can't the "many somethings" just be an infinite list itself?)

Realtime input, like that from joysticks, mice, keyboards, and http sockets, spring from a Configure Hardware function. That function returns an infinite list (so, use lazily) of all the input that stream will ever get. That list must be passed around as normal, or put into a "global" variable, which passes it implicitly. (Does it return it implicity as well?)

Namespaces are Named Contexts, and Have Properties Which Are Functions.

It is said that most people use only about 2,000 words in their everyday vocabulary. My experience with Inform 7 has taught me that because of this fact, namespaces are an important addition to English-like programming languages. (One plea I made on the Inform 7 forums, directed to the creators of libraries and extensions, was to "not use up all the good words".) The ordinary word for namespaces is context, and deciding what is or isn't exported outside the namespace is the primary purpose: FOOBAR ONLY PROVIDES GIVING A CAR TO A PERSON, DRAWING A SKILL, AND CARRIAGES. And to use other namespaces, [THIS] FOOBAR ONLY NEEDS FOO, AND BAR. Usually languages require a keyword on each method or variable, but Complish wraps it all up into a single sentence. Or two or three, one for each level of access defined.

Complish can take context a bit further than just the definitions of words. Many properties of how things work under the hood, what some languages call a metaobject protocol, allow rewriting and rewiring such basics as calling conventions, rounding floats back to ints, growable ints, closure variable capture, inheritance, functional purity, automatic decorators like backtracking instrumentation, member accessibility, parsing of the source itself via a parse/print function on a preexisting user-defined object, etc. A context has properties which are functions, with the default implementation of how these glue bits work. Assign a different function of the same (or contravariant) signature to these properties to alter how the code in the context is put together.

Object Initializers.

An object initializer looks like FIDO HAS A COAT OF BROWN, A BUOYANT PERSONALITY, AND [AN AGE OF] THREE YEARS. If only one property's type accepts a value, then the property needn't be named: the value three years is enough. Similar for enum properties like booleans or even personality, if we had worded personality as nouns instead of adjectives. Otherwise, the construct "A (PROPERTY) OF (VALUE)" works.

The construct FIDO IS BUOYANT only works if Complish already knows buoyant is an adjective and/or Fido is a struct.

The rules about uniquely typed values in object creation apply to complex instances as well: TOBY HAS A BUOYANT PERSONALITY AND A 1978 CHEVROLET MUSTANG. (But can any object be named according to its values? In other words, if the totality of an object's values is in the name, do we really need a "name"? Esp. if closures' read-only nature comes into play.)

If regexes are part of the compiler, use them to describe how to parse a struct instance, which is trickier than how to write one.

Business Programming Includes Videogame Programming: Lessons from PS3 and XBox 360 Devs.

- **Functional programming is the right default for multicore cpus, but imperative constructs are vital.** (multimethods intended to be pure functional, AFTER rules can do I/O because they can see into the multimethod. Rules cannot affect multimethods.) Perhaps disallowing globals... but that still doesn't allow objects to have member variables. But if member variables can be thought of as a shortcut to writing them out in every method call in the class, and the constructor virtually passes them in to itself with their initial values, then the function constructing the object is passing those in a ordinary arguments. So, all member vars must be

initialized from the scope of the caller of the constructor, not from the in-object scope? Also, but if multimethods don't belong to objects, and the member variables belong to the caller of the constructor (allocated on that function's call stack), then what's left? Objects and closures are the same, this closure is a nil closure – calling it doesn't do anything because no methods belong to it – closures are usually curried until only a single return value remains. So can these objects "shrink" to nothingness? Then what is Type?

- **Imperative constructs are vital. Wrap them in dedicated "side-effect" Monad types?** If so, then define partiality as a Effect, thus the base language is amendable to proofs.
- **Lenient evaluation is better than eager or lazy evaluation.** Eager eval is an optimization. If so, support lazy eval with explicit keywords? Infinite lists etc. should always be lazily evaluated of course.
- **Features that aid reliability also enable concurrency.**
- **Mutable state & heaps are OK within a function, if used locally & destroyed before function's end.**
- **Don't tie Nullable to reference-type/value-type distinction. If a type isn't explicitly nullable, then it isn't null.**
- How is the bottom type written in Complish? Can Nothing be a determiner, replacing a/many with "no"? Sure. But how do you specify a nullable X as a parameter distinct from "an X"? The adjective possible or optional? The word maybe?
- Covariance means $\text{OBJECT X} = \text{NEW STRING}()$; works, as you can put a more derived ("smaller") type into a spot intended for a less derived ("larger") type. Contravariance is used on function types as you can put a larger, base class type into a spot for a smaller, more derived type. This is because a function variable that can hold a cat-accepting function could also hold an animal-accepting function, because all the pre-existing invocations of that function-within use cats, or more specific kinds of cats. Since the animal-accepting function also accepts cats, we can put the animal-accepting function into that variable. IEnumerable is declared as IEnumerable<out T> so it can accept covariant values: an IEnumerable<object> accepts a list of strings.

Miscellany.

"Real" functional programming is referential transparency because of immutable aggregate-values, not function composition. I like referential transparency partly because there's no difference between a value and a reference to a value. But ref trans works by making everything work like value types. What if we made everything work like reference types? Well, i wouldn't know if a global var is supposed to be a reference to an existing item or is a new item altogether.

Regexes and Prolog both use backtracking.

Regexes can transform a string into a boolean or a propertied object.

Can method chains backtrack?

Method chaining with branches is dataflow programming. (Minor note: multiple data items are in multiple stages simultaneously during dataflow programming, but without necessarily being concurrent.)

Closures are defined imperatively; objects are defined declaratively.

Declarative statements are ALWAYS; imperative statements are AFTER previous imperative statements.

Aspect-oriented rules and functional programming complement one another: one factors out side-effects while the other disallows side-effects.

When user code uses string.ToInt(), it is effectively calling a part of the compiler.

Method chaining is main-verb chaining.

A class can have properties itself, separate from the properties on its instances. THE TEXTUAL SERVICE OF EVERY BASEXO IS "FOO".

This isn't to be confused with with defining an enum property, as THE MODEL OF A CHEVY IS EITHER MUSTANG OR CAMRY.

I want an easy way of splitting an object's properties into smaller objects, or creating larger objects via the properties of smaller objects. A lot of business programming is this.

A business rule is frequently a constraint on a web of objects.

It should always be possible to delete an item from the list while iterating through the list.

I want static object methods that can be inherited, and I want properties on the classes themselves (metaclass properties, to be precise) which inherit and can be initialized at compile-time or at run-time. (Like C# READONLY.) I want this for ContactXO.Get(lambda) to return an item, or null, the latter of which constructors can't do.

Hypernym/hyponym = superclass/subclass. Holonym/meronym = container/item or whole/part.

Writing in imperative style isn't natural because people don't talk in long lists of imperatives. Writing in functional style isn't natural because it's endlessly digressive. But people do speak in a story style which is still chronological.

Implicit Loops?

A list of statements ("many statements") can also perform a chain of transforms. "For" is a preposition, and since prepositions herald a parameter, Complish allows adding "for <many X>" to any invocation which doesn't already use "for" for its own purposes. The parameter which is being iterated over is "each X", but perhaps that parameter can be dropped when "for" comes to visit.

Generics.

A type can be passed to a function, but usually when that's done it is considered partial application: the other parameters have arguments, and the one(s) that got the type don't yet, so the resulting function still accepts that parameter.

There's no distinction between returned values (output) and arguments to parameters (input) in Complish, because a relation can be "called" in multiple combinations.

If at least one of those params/returns isn't generic, then in a referentially transparent language, there's no possible way for the function/relation to be generic.

So, syntax: how to specify a generic function? How about the function's declaration has a parameter, that looks like most of the parameters, be of type TYPE? All parameters of type TYPE are type-parameters, not actual parameters. Can any meaningful constraint be placed on it?

The best use case for generics, list of X, English already has special syntax for via determiners. Visual Basic uses "of <type>" for generics, though Complish already uses "of" for properties in "of <objvar>".

From the Ground Up.

BNF Grammar for Complish:

SINGULAR ARTICLE ::= A | AN

MASS NOUN ARTICLE ::= SOME

LIST ARTICLE ::= MANY | SEVERAL

ARTICLE ::= <SINGULAR ARTICLE> | <MASS NOUN ARTICLE>

DETERMINER ::= <A> | <LIST ARTICLE>

PREPOSITION ::= // there are over 300 of these //

LIST1<X> ::= (<X> ,)* (<X> ,? AND)? <X>

LIST2<X> ::= (<X> ;)* (<X> ;? AND)? <X>

LIST<X> ::= LIST1<X> | LIST2<X>

BASICTYPE ::= NUMBER | TEXT | BOOL | MONEY | TIME | ORDINAL | PERCENT | TYPE

OBJECTTYPE ::= <ARTICLE> <NEW NAME NOUN> IS ALWAYS? <LIST<SLOT>>

OBJVAR ::= THE <OBJECTTYPE>

TYPE ::= BASICTYPE | OBJECTTYPE

VAR ::= THE <TYPE>

NULL BOTTOM ::= NO <TYPE>

PROPERTY ::= THE <SLOT.NAME> OF THE <OBJVAR>

EXPRESSION ::= <MATH> | <BOOL-ALGEBRA> | <PROPERTY> | <RELATIVE CLAUSE> | <LITERAL> | <VAR>

NOUN ::= <EXPRESSION>

PREPOSITIONAL SLOT ::= <PREPOSITION> <SLOT>

FUNCTION BODY ::= PREPOSITIONAL SLOT wherein PREPOSITION = BY | VIA and SLOT = <LIST<INVOCATION>>

Note: one function body per relation please.

RELATION ::= <SLOT>? <NEW NAME VERB> <SLOT>? <PREPOSITIONAL SLOT>*

Note: comma'd slots are accessed via property syntax. Preposition'd slots are accessed by parameter auto-naming syntax. If the verb is also a noun, like TO PURCHASE, then either syntax works nicely.

FUNCTION ::= RELATION wherein one slot is a FUNCTION BODY

STRUCT ::= RELATION wherein no slots are a FUNCTION BODY

INVOCATION ::= <NOUN>? <VERB> <NOUN>? (<PREPOSITION> <NOUN>)*

PAST PARTICIPLE ::= <FUNCTION> wherein one type is inputted and one type is outputted and the verbform is past tense

SLOT ::= <DETERMINER> (<PAST PARTICIPLE>)* (<NEW NAME ADJECTIVE>? <TYPE.NOUNNAME> | <TYPE.ADJECTIVENAME> <NEW NAME NOUN>)

RELATIVE CLAUSE ::= THE <SLOT.TYPE> WHICH <INVOCATION SANS THAT SLOT>

TEMPORAL CONJUNCTION ::= BEFORE | AFTER

TEMPORAL CONSTRAINT ::= <RELATION DESCRIPTION> <TEMPORAL CONJUNCTION> <RELATION DESCRIPTION>

INVARIANT CONSTRAINT ::= <EXPRESSION> IS ALWAYS? <EXPRESSION>

FUNCTION BODY EXTENDED ::= FUNCTION . (<INVOCATION> .)*

SENTENCE ::= (<RELATION> | <OBJECTTYPE> | <INVOCATION> | <TEMPORAL CONSTRAINT> | <INVARIANT CONSTRAINT>) .

RELATION DESCRIPTION ::= ?like Zarfian?

.

.